informalities such as found on pages 10 and 17. It is respectfully submitted that the Examiner's indication that "betch" on page 13, line 28 be "batch" is incorrect, rather the correct term is fetch. Respectfully, no new matter is being submitted by these corrections to the specification.

Further, Applicants take this opportunity to amend the Abstract and clarify that after the system tries to locate a linear path from the beginning of the method to the destination program counter, it will iteratively process the sequence of bytes at each branch until the destination program counter is reached. The term "interactively" in the Abstract as originally filed is being deleted and replaced by the correct term iteratively. Respectfully, no new matter is being submitted by this correction to the Abstract as it is clear from the specification and drawing Figures 1A and 1B the iterative process steps 104-108.

Next, in the Official Action, the rejected Claims 1-3,10-13, and 20-23 under 35 U.S.C. §102(e) as allegedly being unpatentable over Agesen (U.S. Patent No. 6,047,125). The Examiner further rejected Claims 4-5 and 14-15 under 35 U.S.C. §103(a) as allegedly being unpatentable over Agesen (U.S. Patent No. 6,047,125) in view of Agesen (U.S. Patent No. 5,909,579). The Examiner then rejected Claims 6-7, 16-17 and 19 under 35 U.S.C. §103(a) as allegedly being unpatentable over Agesen (U.S. Patent No. 6,047,125) in view of Gosling (U.S. Patent No. 5,668,999). The Examiner further rejected Claims 8 and 18 under 35 U.S.C. §103(a) as allegedly being unpatentable over Agesen (U.S. Patent No. 6,047,125) in view of Gosling and further in view of O'Connor (U.S. Patent No. 6,098,089).

With respect to the Examiner's rejection of Claims 1-3,10-13, and 20-23 under 35 U.S.C. §102(e) as allegedly being unpatentable over Agesen (U.S. Patent No. 6,047,125), Applicants respectfully disagree. The present invention is directed to a method for mapping a stack in a stack machine environment to help determine the shape of the stack at a given program counter. As now set forth in amended Claims 1 and 11, this is accomplished by locating all start points possible for a given method, that is, at all of the entry points for the method and all of the exception entry points, and trying to find a path from the beginning of the method to the program counter in question, and then iteratively processes the sequence of bytes at each branch until the destination program counter is reached. Once the path is found, a simulation is run of the stack through that path, which is used as the virtual stack for the purposes of the garbage collector.

It is respectfully submitted that the Agesen '125 patent does not anticipate the invention as set forth in amended Claims 1 and 11. Agesen 1125 essentially is directed to a method for modifying a sequence of instructions to improve memory management within a storage device during execution of the instructions, which comprises steps, performed by a processor, of (a) analyzing the sequence of instructions for a conflict indicating an undeterminable variable type, (b) determining the type of conflict, and (c) modifying the sequence of instructions to eliminate the conflict based on the determination. Further, as stated in the Summary (Col. 5, lines 16 et seq. of Agesen '125 patent) the subject matter of Agesen is to improve garbage collection by modifying code that introduces a conflict in the assignment of variables for identifying references. The conflict exists, for example, when a method includes code defining at least two control paths leading to a common subroutine and the garbage collector initiated during execution of the subroutine cannot determine whether a variable represents a reference. By rewriting the code in accordance with the principles of the present invention to eliminate the conflicts, the collector is able to reclaim memory space effectively during execution of the subroutine.

Thus, it appears that the substance of the invention in Agesen '125 is to modify the program bytecodes in the stack, Col. 5, lines 23-25. Every embodiment of Agesen '125 seems to include the step of rewriting some code in the program. The present invention does not involve any rewriting of code at all, and the Claims 1 and 11 of the present invention are not directed as such. Rather, the present invention is directed to an efficient and complete stack walking method, rather than the code repair method as described in Agesen '125. Further, the present invention deals with the dynamic nature of the stack by leaving open a small area of memory in the stack and tagging it for dynamic mapping. Agesen '125 does not do that at all.

As to the Examiner's reference to the subject matter of Agesen '125 at Col. 11, lines 59-63, again the code in Figure 7 is an example of what bytecodes the invention might encounter in code created by a Java compiler (see Col. 10, lines 49-52 of Agesen '125). In fact, the Examiner is respectfully requested to refer to Agesen '125 at Col. 11, lines 64-65, where it describes the the conflicts that the Agesen '125 invention is intended to resolve. The same is true for Col. 7, lines 1-5. The problem he describes there is what his invention sets out to solve.

The Examiner's rejection of claims 2, 3, 12 and 13 is similar. Agesen '125 teaches rewriting the Java code, not walking the stack. At no point is the bytecode rewritten in the present invention. Agesen '125 is fundamentally about rewriting bytecodes to enable accurate garbage collection, whereas the present invention is directed to taking bytecodes and inferring stack shapes from it. The Examiner's reference to Col. 7, lines 42-44 and to Col. 11, lines 54-59 and other references indicates what a standard computer system does, not to what Agesen's invention is directed to.

Respectfully, the Examiner has not shown an anticipation, and it would appear that his rejection of claims 1-3, 10-13 and 20-23 is not supportable. In view of the foregoing, Applicant respectfully requests the Examiner to withdraw the rejection of Claims 1 and 11 under 35 U.S.C. §102(e) and the rejection of Claims 2-3, 10 and 12-13 and 20-23 by virtue of their dependency.

With respect to the Examiner's rejection of Claims 4-5 and 14-15 under 35 U.S.C. §103(a) as allegedly being unpatentable over Agesen '125 in view of Agesen '579, the Applicants respectfully disagree. Agesen '579 is directed to a system that chooses to store full information for only every *n*th bytecode in the stream, and recreating the deltas in between them (See Abstract). In view of the arguments above, and by virtue of their dependency upon amended Claims 1 and 11, Claims 4-5 and 14-15 are patentable over the combination of Agesen '125 and '579 and the Examiner is respectfully requested to withdraw the rejection of Claims 4-5, 14-15 under 35 U.S.C. §103(a).

Furthermore, with regard to Claims 6 and 16, the Examiner states that Gosling supplies at Col. 5, lines 21-29, the step of generating a virtual stack that is missing from Agesen '125. However, Claims 1 and 11, from which Claims 6 and 16 respectively depend, are restricted to mapping a path of control flow only from an arbitrary start point up to a destination program counter. Therefore, the virtual stack thus created is only a stack for part of the program, unlike Gosling '999 which creates a virtual stack for the entire program "used in the same way as a regular stack", see Gosling at Col. 5, lines 35-40. Thus, Claims 6 and 16, depending from claims 1 and 11 respectively, have not been shown to be obvious over the cited art and should be allowed. By virtue of their dependency, Claims 7 and 17 are likewise allowable for the reasons discussed herein.

Regarding the rejection of Claims 9 and 19, depending from Claims 7 and 17 respectively, the storing of a bitstring to a pre-allocated area on the stack is described at page 19, lines 1-4 of the present specification. Although storing information to a pre-allocated area on the stack is known in general, the claimed pre-allocated area is used when a special event occurs. In contrast, Agesen '125 stores the value at a position selected not by the invention but by the compiler. As stated at Agesent '125 Col. 11, lines 9-11: "The compiler selected variable 3 to store the value for itmp so bytecode 704 stores the value (1) from the top of the operand stack in variable 3." This storage is simply matching the storage location to the location chosen by the compiler.

Further with respect to the rejection of Claim 9, the pre-allocated area on the stack is distinguishable from that of Agesen '125 as indicated by the Examiner at Col. 11, lines 5-12. Respectfully, Agesen '125 is describing what happens in the running program, not where to store stack-map data, which the running program has no knowledge of. Effectively, in the present invention, stack space is being reserved so that the computed stack maps may be placed when the program analysis is finished.

In view of the arguments above, and by virtue of their dependency upon amended Claims 1 and 11, Claims 6-7, 16-17 and 19 are patentable over the combination of Agesen '125 and Gosling and the Examiner is respectfully requested to withdraw the rejection of Claims 6-7, 16-17 and 19 under 35 U.S.C. §103(a).

With respect to the rejection of Claims 8 and 18 over Agesen '125 in view of Gosling and O'Connor, the Examiner states that although neither Agesen '125 nor Gosling teaches storing the bitstring on a heap, O'Connor states that it is well known in the art that "garbage collection" of "heap-allocated storage" is an "attractive model for dynamic memory management", Col. 1, lines 39-42 of Gosling. While it certainly is desirable to do garbage collection, there are many challenges around how to do it effectively and efficiently. Indeed, the present specification reveals this at page 2, lines 9-14. In distinguishing Claims 8 and 18 from what O'Connor seems to be stating, it is respectfully submitted that Claims 8 and 18 depend from Claims 7 and 17 respectively, whose allowability has been discussed herein. Claims 8 and 18 add the limitation that the step of storing the bitstring comprises storing the bitstring to the selected method as compiled on a heap. The related description is at page 17, line 23 to page 18, line 9 of the present specification. At lines 1-2, the description states: "the compiled method also includes a field for the stack map 412." The cited passage from O'Connor does not indicate anything about the stack map being placed in the heap within the compiled method. In general, O'Connor discusses ordinary stores to the heap but not the storage of a stack map.

In view of the arguments above, and by virtue of their dependency upon amended Claims 1 and 11, Claims 8 and 18 are patentable over the combination of Agesen '125, Gosling and O'Connor, and the Examiner is respectfully requested to withdraw the rejection of Claims 8 and 18 under 35 U.S.C. §103(a).

Attached hereto is a marked-up version of the changes made to the specification by the current amendment.

In view of the foregoing, Applicant believes that this application is in condition for allowance and Applicants henceforth respectfully solicit such allowance. If the Examiner believes a telephone conference might expedite the prosecution of this case, Applicants respectfully request the Examiner to call the undersigned, Applicants' attorney, at: (516) 742-4343.

Respectfully submitted,

[illegible]

Steven Fischman
Registration No. 34,594

SCULLY, SCOTT, MURPHY & PRESSER
400 Garden City Plaza
Garden City, New York 11530
(516) 742-4343
SF:gc

Serial No.: 09/329,558 **Docket:** CA919980012US1 (12463)

MARKED-UP VERSION OF THE CHANGES MADE

IN THE SPECIFICATION:

Please replace the paragraph on page 2, lines 6-14 with the following:

-- The effective management of dynamic memory, to locate useable free blocks and to deallocate blocks no longer needed in an executing program, has become an important [programming] programming consideration. A number of interpreted OO programming languages such as Smalltalk, Java and Lisp employ an implicit form of memory management, often referred to as garbage collection, to designate memory as "free" when it is no longer needed for its current allocation.--

Please replace the paragraph on page 10, lines 4-14 with the following:

--Returning to Figure 1A, the input to the method of the invention is the destination PC for the method and the storage area destination to which the resulting information on the stack shape will be written (block 100). When the mapping occurs at runtime, the definition of the storage destination will point to a location on the stack; when the mapping occurs at compile time, the pointer will be into an array for storage with the compiled method on the heap. The different uses of the invention for stack mapping at runtime and at compilation are discussed in greater detail below.--

Please replace the paragraph on page 13, line 28 through page 14, line 4 with the following:

--Temporary [betch] fetch and store instructions are normally one or two bytes long. One byte is for the bytecode and one byte is for the parameter unless it is inferred by the bytecode. However, Java includes an escape sequence which sets the parameters for the following bytecode as larger than normal (wide bytecode). This affects the stack mapper only in how much the walk count is incremented for the next byte. It does not affect control.--

Please replace the paragraph on page 17, line 23 through page 18, line 9 with the following:

--The compressed encoded stack map is stored statically in the compiled method or on the stack during dynamic mapping. In the case of static mapping, a stack map is generated and stored as the method is compiled on the heap. A typical compiled method shape for a Java method is illustrated schematically in Figure 4. The compiled method is made up of a number of fields, each four bytes in length, including the object header 400, bytecodes 402, start PC 404, class pointers 406, selector 408, Java flags 410 and [laterals] literals 414. According to the invention, the compiled method also includes a field for the stack map 412. The stack map field 412 includes an array that encodes the information about the temps or local variables in the method generated by the stack mapper in the manner described above, and a linear stack map list that a garbage collector can use to access the stack shape for a given destination PC[ ] in the array by calculating the offset and locating the mapping bits in memory.

IN THE ABSTRACT:

Please amend the Abstract of the Invention as follows:

--The stack mapper of the present invention seeks to determine the shape of the stack at a given program counter. This is accomplished by locating all start points possible for a given method, that is, at all of the entry points for the method and all of the exception entry points, and trying to find a path from the beginning of the method to the program counter in question. The mapper first tries to locate a linear path from the beginning of the method, and then [interactively] iteratively processes the sequence of bytes at each branch until the destination program counter is reached. Once the path is found, a simulation is run of the stack through that path, which is used as the virtual stack for the purposes of the garbage collector.

IN THE CLAIMS:

Please amend Claims 1 and 11 as follows:

1. (Amended) A method for mapping a valid stack up to a destination program counter, comprising:

mapping a path of control flow on the stack from any start point in a selected method to the destination program counter by locating a linear path from the beginning of the method to the destination program counter and iteratively processing a bytecode sequence for each branch until said destination program counter is reached; and

simulating stack actions for executing bytecodes along said path.

11. (Amended) A method for mapping a Java bytecode stack up to a destination program counter comprising:

mapping a path of control flow on the stack from any start point in a selected method to the destination program counter by locating a linear path from the beginning of the method to the destination program counter and iteratively processing a bytecode sequence at each branch until said destination counter is reached; and

simulating stack actions for executing bytecodes along said path.